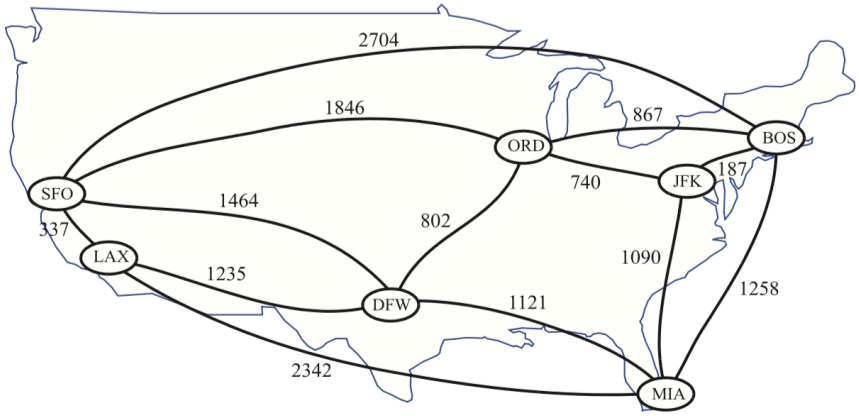
CSE221 Data Structures (Fall 2016)
Instructor: Prof. Tsz-Chiu Au
Due date: Dec 4, 2016, 11:59 pm.

Assignment 5: Directed Graphs and Dijkstra's Algorithm

Many problems in computer science, especially Artificial Intelligence, involve finding the shortest paths in a directed graph. How to correctly implement the shortest path algorithms such as the famous Dijkstra's algorithm is an essential skill for computer science students. The performance of these algorithms heavily depends on the choice of container classes for storing directed graphs. In this exercise, you will learn how to implement the adjacency list structure for directed graphs and Dijkstra's algorithm for solving the single-source, shortest-path problems. At this end, you will write a program to help travelers to solve their shortest path problems they encounter in real life.

Our Goal: Finding the Shortest Routes in a Flight Map

Let's take a look at the following map in Figure 13.13 of the textbook:



This figure shows a map of the flight connections in the major airports in the United States. As you can see, there are some airports that do not have direct flights between them. Hence, a passenger would have to choose a route with multiple connecting flights to get to his destination if there is no direct flight to his destination. The question for the passenger is: what is the shortest route to reach the destination? This is the classic scenario for the study of the single-source, shortest-paths problem in graph theory. In this exercise, you will have to write a program to help the passenger to solve this problem. Your program will load the data of flight connections from a file and run the Dijkstra's algorithm to find the shortest routes between two airports.

Implementing Adjacent Lists of Directed Graphs

While the example in Figure 13.13 assumes the graph of the flight connections is undirected, a more realistic assumption is that the flight map is a directed graph in which the flight distance of the two directions between two airports can be quite different. While there are many ways to implement an adjacency list structure, we want you to implement the one as described in Section 13.2.2 in the textbook, as its operations can achieve the running times in Table 13.2 of the textbook. To achieve the running time of O(1) of `eraseEge()` and O(deg(v)) of `eraseVertex()`, it is necessary to maintain the positions (i.e., iterators) of the collections of vertices and edges in the graph. Other simple implementations of adjacency list structures without maintaining these positions may not guarantee the time complexity of the above operations.

In this exercise, you will implement the adjacency list structure as described in Sections 13.1.1, 13.2.2, and 13.4 of our textbook to store the directed graph in the flight map. Your implementation should faithfully follow the description in the textbook. On Page 599, the textbook has provided all the details the interface of a Graph ADT, which includes the following member functions: `Vertex::operator*()`, `Vertex::incidentEdges()`, `Vertex::isAdjacentTo()`, `Edge::operator*()`, `Edge::endVertices()`, `Edge::opposite()`, `Edge::isAdjacentTo()`, `Edge::isIncidentOn()`, `vertices()`, `edges()`, `insertVertex()`, `eraseVertex()`, and `eraseEdge()`. Since we will implement a directed graph, the Graph ADT should also include the following member functions as described on Page 626: `Edge::isDirected()`, `Edge::origin()`, `Edge::dest()`, and `insertDirectedEdge()`. Notice that we will omit `insertEdge()` as it is replaced by `insertDirectedEdge()` as discussed in Section 13.4.

In addition, you need to implement five more member functions for directed graphs:

- `Vertex::isOutgoingTo(v)`

  This function returns true if and only if there is a directed edge connecting the current vertex to the given vertex v.

- `Vertex::outgoingEdge(v)`

  This function returns the directed edge connecting the current vertex to the given vertex v. If there is no such edge, it throws an exception.

- `Vertex::outgoingEdges()`

  This function returns the set of all directed edges connecting the current vertex to any vertex in the graph.

- `Vertex::operator==(v)`

  This function returns true if and only if the current vertex is the same as the given vertex v.

- `Edge::operator==(e)`

  This function returns true if and only if the current edge is the same as the given edge v.

We provide you a skeleton of the `AdjacencyListDirectedGraph` class in `AdjacencyListDirectedGraph.h`, which contains the headers of all of the above member functions. All you need is do is to implement these member functions according to the adjacency list structure in the textbook.

An overview of the adjacency list structures is shown in Figure 13.4 on Page 603 in Section 13.2.2. Your implementation should exactly implement the structure in this figure. Notice that this adjacency list structure is built on top of the edge list structure in Section 13.2.1, which means that you will have to understand the edge list structure as well. To facilitate a correct understanding of the implementation, we have already defined the classes for vertex objects and edge objects in `AdjacencyListDirectedGraph.h` for you. You just need to take a look at the classes to see how they correspond to the description in the textbook. The classes have included all the member variables as described in Section 13.2.

In the code we provide in `AdjacencyListDirectedGraph.h`, all vertex objects, edge objects, and incidence collections are stored in these member variables: `vertex_collection`, `edge_collection`, and `inc_edges_collection`, which are `list` objects in STL. These are all the member variables you need; and you should not declare any other member variables, even a private one, in the `AdjacencyListDirectedGraph` class. Notice that you cannot use vectors or some other container classes to store the vertex objects, edge objects, and incidence collections, because their iterators will become invalid after you modify the containers. The `list` container class in STL is one of the few that does guarantee that its iterator will continue to point to the same objects after you modify the container.

In this exercise, all you need to do is to implement the member functions in `AdjacencyListDirectedGraph`. You should not modify any code that are given in `AdjacencyListDirectedGraph.h`, and you should not add any other member variables and member functions. To simplify the implementation, you are allowed to assume that vertices and edges passed as arguments to the member functions are never "NULL" (i.e., they always refer to a vertex object or an edge object). This assumption can save you some codes for error handling. There is also no need to define the constructor and the destructor as the list member variables will automatically "free" their own memory.

Please make sure that your code can be compiled using the gcc compiler on our submission server. Please write your name, your student ID, and your email as a comment at the top of the file. You should also briefly describe your implementation at the top of the file.

Implementing Five Member Functions of `FlightMap` in `assignment5.cpp`

We also provide you `FlightMap.h`, in which we define the `FlightMap` class that will be used in `main.cpp`. The `FlightMap` class contains all the functionalities this are needed in our application—there is no need to extend it to add more public member functions. However, you are allowed to add some private member functions if you need (see below).

For each member function of the `FlightMap` class, we have written a detailed description of its inputs, its outputs, and its function as the comment in `FlightMap.h`. Please take a look at the comments to learn what these member functions are. We have also implemented most of the member functions in `FlightMap.cpp`. We strongly recommend you to read the codes in `FlightMap.cpp` to learn about how to use the `AdjacencyListDirectedGraph` class.

There are five member functions of the `FlightMap` class that are not implemented in `FlightMap.cpp`:

- `calcRouteDistance(route)`

  Calculate the total distance of a route.

- `findRoute(airport1, airport2)`

Find a route between two airports, which must be a simple path with no cycle.

- `findReachableAirports(airport)`

  Find the set of all airports reachable from an airport.

- `findShortestRoute(airport1, airport2)`

  Find the shortest route between two airports.

- `printAllShortestRoutes(airport)`

  Print all shortest routes to all airports reachable from a given airport.

You will have to implement these member functions in `assignment5.cpp` (not in `FlightMap.cpp`). Once again, the comments in `FlightMap.h` provide a detailed description of the inputs, the outputs, and the functions of these member functions. You should also take a look at the outputs of the sample program called `flight-map-sample` to see exactly how they work.

You are allowed to include some private member functions to help you implement the above member functions. You will have to declare them in `FlightMap.h`, and submit the file along with `assignment5.cpp`. However, we insist that you should not modify the existing codes in `FlightMap.h`. You should not add any member variables and public member functions in `FlightMap.h`. Please take a look at `FlightMap.h` and locate the line: "You should not modify anything above this line in this class."

Implementing Dijkstra's Algorithm

In `FlightMap::findShortestRoute()` and `FlightMap::printAllShortestRoutes()`, you will have to use a shortest path algorithm to find the shortest routes. In this exercise, you will implement Dijkstra's algorithm for finding the single-source, shortest-paths in a directed graph. Section 13.5.2 in the textbook is dedicated to the discussion of the algorithm. More specifically, The Code Fragment 13.24 is the pseudo-code of Dijkstra's algorithm. Please read the pseudo-code to learn how the algorithm works. While the discussion in Section 13.5.2 is for undirected graphs, the same algorithm will work for directed graph with very little modification.

In fact, there are many different ways to implement Dijkstra's algorithm, and you are free to explore other options. In addition, instead of implementing your own priority queue, we will allow you to use `priority_queue` in STL to implement the

algorithm. However, there is one drawback of `priority_queue` in STL: it does not support the modification of a key of an element in a priority queue, and you cannot remove any non-top element from the priority queue. Dijkstra's algorithm requires the modification of keys in the priority queue in the relaxation step. Fortunately, there is an implementation of Dijkstra's algorithm that does not require "modifying" non-top elements. The idea is that a vertex can be inserted into a priority queue *multiple* times whenever its cost is getting lower, without removing any one of them from the priority queue. The algorithm also maintains the set of vertices that have been "visited" (i.e., their shortest paths have been found). When popping from the priority queue, the algorithm ignores the vertices that are "visited". The effect will be the same as modifying the keys of non-top elements in the priority queue.

There is a high chance that you will use `list`, `vector`, and `priority_queue` in STL to implement your algorithm. We expect you to learn how to use these STL classes yourself by reading some online materials. Notice that the C++ compiler on our submission server supports a rather old version of C++ compiler only. Hence, you should probably need to use the features of C++98 of these STL classes only.

Testing and Submission

We provide you a sample program called "`flight-map-sample`" that implements all functionalities of this program. We also provide you a graph data file called `graph1.txt`, which contains the data of the graph in Figure 13.15 of the textbook. Please run the sample program with the graph data file on our submission server to see the output it generates. Please make sure that the output of your program is the same as the output of the sample program (except the white spaces and the error messages in the exceptions).

You should check the correctness of Dijkstra's algorithm in your program by comparing your solution with the one in Figure 13.15. Apart from `graph1.txt`, we recommend you to write a few more graph data files of your own to test your program.

In this exercise, you are allowed to use four container classes in the Standard Template Library: `list`, `vector`, `queue`, and `map`. However, you should not use other data structures in STL except the iterators of these container classes and some common exceptions such as `runtime_error`. If in doubt, please contact the instructor to ask whether a data structure in STL can be used.

We will test your implementation of `AdjacencyListDirectedGraph` using a different main function that is different from the one in main.cpp. We will test the completeness and correctness of your implementation, including whether your code

will throw all necessary exceptions. We will also check whether your program will cause memory leak. We will also test whether `assignment5.cpp` uses the member functions in `AdjacencyListDirectedGraph` correctly by replacing `AdjacencyListDirectedGraph.h` with the instructor's implementation of `AdjacencyListDirectedGraph.h`.

The error messages in `runtime_error` exceptions do not have to be exactly the same as the messages in the sample program we provided. You can write the error messages in your own sentences. However, we will check whether you have thrown the exceptions.

To compile your program on our submission servers, use this command:

```
g++ -o flight-map FlightMap.cpp assignment5.cpp main.cpp
```

Before you submit your program, please check whether your program can be compiled correctly using this command on our submission server. We do not grade your program using other compilers.

Please also submit a plain text file called "`README.txt`" to tell us the extent of your implementation, more specifically which parts have been implemented and which parts have not. Also, if there are some known bugs in your program, you should state them in the plain text file.

You will submit only four files: (1) `AdjacencyListDirectedGraph.h`, (2) `FlightMap.h`, (3) `assignment3.cpp`, and (4) `README.txt`. These files should be self-contained and you cannot submit additional files. You should not submit `main.cpp` and `FlightMap.cpp` as you will not modify them. Please put these files in a zip file called `assign5.zip`, and submit it using the `dssubmit` script on our submission servers (uni06~10.unist.ac.kr). The submission command is

```
dssubmit assign5 assign5.zip
```

Bug Reports

Please report any bugs in the codes we provide as well as any typos and errors in this handout to **Prof. Tsz-Chiu Au** at **chiu@unist.ac.kr**. We will look into the bug reports and fix the problems. If we have to release a new version of the codes to fix the bugs, we will announce it on our course webpage or Blackboard. Before you submit your program, you should check the announcement to see whether the codes have been updated. Please make sure that your program works with the final version of the codes we provide.